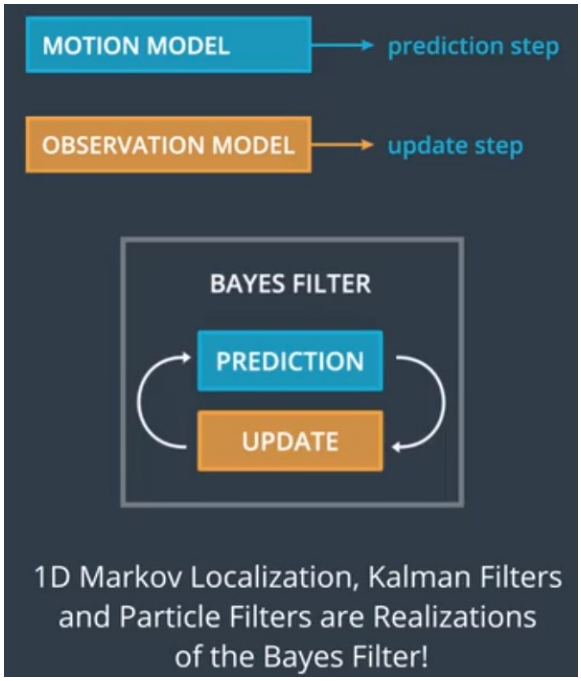
Landmark-Based Localization

Step 0: Decide on the approach

My knowledge and intuition took me directly to explore Bayes (Localization) Filter. Bayes Filter is a general framework for recursive state estimation.
Can be considered as an algorithm used for calculating the probabilities of multiple beliefs to allow a robot to infer its position and orientation. Essentially, Bayes filters allow robots to continuously update their most likely position within a coordinate system, based on the most recently acquired sensor data.
It consists of two parts: prediction and update.



The first step I took was to decide which specific “realization” would fit better this problem.

Kalman Filter (KF)

For the basic Kalman filter there are three assumptions:

- The underlying system is a linear dynamical system
- All error terms and measurement noises are white
- All error terms and measurements have Gaussian distributions

The first assumption/requirement was not met in this problem. The 2-wheel differential drive model used in this project is non-liner. Also both of the measurement/observation models, the landmark sensing model and the odometry model, are not linear.
For these reasons, the next possibility was to use an Extended Kalman filter (EKF).

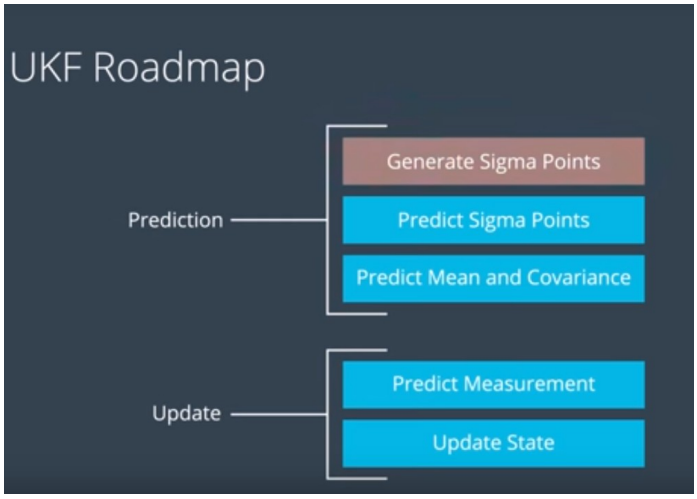
Extended Kalman filter (EKF)

A common approach to develop an EKF is to make a linear model approximation of the nonlinear system. However, this can

cause loss of important information, that otherwise require many additional state variables to keep nearly the same accuracy as the corresponding nonlinear model. A better solution will be to keep the nonlinear model and use that in our algorithm where it is possible. Moreover, when this is not possible, we will make a linear approximation of the model in that specific point of the state space where we are. Calculating the Jacobian of a complex model is both, mathematically tedious to infer and computationally expensive to calculate at every time step. Since EKF applied to complex systems "sticks" the next best thing to explore is the Unscented Kalman Filter (UKF).

Unscented Kalman Filter (UKF)

UKFs allow us to use non-linear systems but they force us to "convert" the multi-modal distributions into Gaussian distributions. Also, UKFs are conceptually complex and harder to implement. Let me show you the UKF implementation Road map.



Each of these steps it's relatively complex on its own, but I won't show you the details since I want to move to my last candidate: Particle filters (PF)

Particle Filters (PF)

Particle filters use a numerical implementation and is, compared to the previous methods, easier to implement, more flexible, multi-modal and can easily handle non-linear and non-Gaussian systems - meaning that the robot can move in e.g. sine-waved motion or the sensor can measure distance with non-Gaussian uncertainties.
For these reasons I decided to proceed with a Particle Filter for this problem. There are Pros and Cons of using particle filters that I'd like to address before I describe the specific approach I took and the details of my implementation.

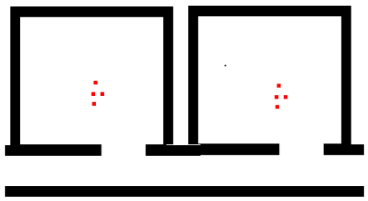
PROS:

- Particle filters can answer most queries by looking at a modest number of samples of Bel(x): This is good because we do not need (and indeed, in most cases would not have) a complete solution of Bel(x).
- The performance of particle filters scale with the amount of resource: Greater computing power leads to more particles, and better results.
- The belief doesn't need any parametric form, as opposed to Kalman filters, which requires Gaussian beliefs

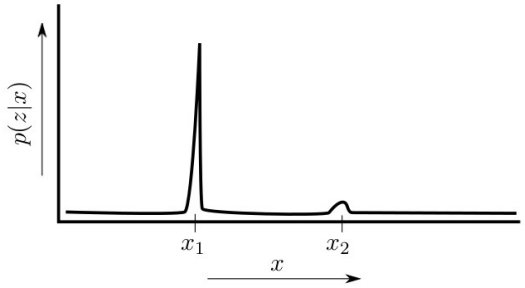
- Works with a finite number of particles: Particle filters (should) use re-sampling to do “survival of the fittest”, thus sampling densely around where the best estimates lie.
- Particle filters are ”easy to implement”, in the sense that the theory is easy to understand
- and put into code.

CONS:

- Repeated re-sampling in the absence of any actual sensory observations could lead to loss of diversity. This problem is illustrated by the following example of a robot localizing itself in two dimensions on a map with two (identical) rooms.



- Really good observation models result in bad particle filters. Look at the 1D example below where the observation likelihood has an accurate and confident (tall and narrow) peak around location x1 , and an erroneous low intensity peak around x2.



The problems here are:

  - Given the narrow width of the peak around x 1 , it is likely that no particle would fall within the region covered by the peak. Thus, the weights of most particles would be driven to zero, and the accurate location estimate (as provided by the observation model) would be lost.
  - It’s likely that a particle (or more) might lie under the region covered by the hump around x2, and its (or their) weight(s) would increase with respect to the other particles, which (as described by the last point) would have infinitesimal weights. This would result in the particle filter providing a confident, albeit completely wrong estimate of the robot’s location.

All these and other problems of Particle Filters can be solved with specific algorithmic techniques and the literature is full of examples.

Let’s jump into how did I try to solve this problem.

STEP 1: Detailed description of the approach selected and its implementation

The robot measures the distance to each landmark and can in this way determine its own position. In theory this is enough to find one unique position of the robot. The problem is though that these

distances are measured with an inaccurate sensor, meaning that no result exists or in other cases only an imprecise result exists.

Implementation details and Particle Filter process

1. **Particle Filter Initialization**: particle_filter.cpp lines 52- 116
   - We collect and make copies of the problem parameters through “RobotParams”.
   - We collect and make copies of all Landmark locations (World Coordinates)
   - We create N particles (robots) around the initial robot state.
   - We add noise to the location and orientation of each particle based on the odometry noise parameters given. It was one of the most difficult points of the problem to be able to apply 4 types of noise in the correct way. Arguably the approach used might not be correct.
2. **Motion Update**: After the robot moves and its odometry sensing package measures this motion, we need to “propagate” this motion to all particles and add noise to it.
   particle_filter.cpp lines 133- 148
   - First we will "move" each particles.
   - As we did earlier, we add noise to the location and orientation of each particle based on the odometry noise parameters given.
   - Finally, we make sure that all the particles are inside the field/map. I decided not to use a cyclic world here.

3. **Sensor Update**: Update particles’ weights and re-sample.
   - **Update Weights**: We update the weights of each particle using a multi-variate Gaussian distribution. You can read more about this distribution here: https://en.wikipedia.org/wiki/Multivariate_normal_distribution
     NOTE: The observations are given in the ROBOT'S coordinate system. The particles are located according to the World coordinate system. We will need to transform between the two systems. Keep in mind that this transformation requires both rotation AND translation (but no scaling). The following is a good resource for the theory: https://www.willamette.edu/~gorr/classes/GeneralGraphics/Transforms/transforms2d.htm and the following is a good resource for the actual equation to implement (look at equation 3.33 http://planning.cs.uiuc.edu/node99.html
     - First we started with a transformation of observations from Particle frame to world frame. The observations are given to us in polar coordinates in the robot frame. We start transforming them to Cartesian coordinates in the robot frame (particle_filter.cpp lines 223- 224). Once all coordinates are in Cartesian, we can now transform from robot/particle reference frame to world frame (particle_filter.cpp lines 227- 229).
     - We move now to the “Association” process, that consist in trying to find which landmark does EACH observation corresponds to. We will simply associate the closest one (desire to closest neighbor algorithm, not enough time). This will actually give the mu(i) on the Weight Update Equation based on The Multivariate-Gaussian probability. First and for efficiency we will go through all the landmarks and make a list of ONLY the ones that are in sensor range (FOV) (particle_filter.cpp lines 246- 266). Then, after that, I will perform a more "computationally heavy" Euclidean distance calculation to narrow down the best landmark-observation association.(particle_filter.cpp lines 276 – landmarkAssociation function)
     - We are finally calculating the Particle's Final Weight. Now that we have done the measurement transformations and associations, we have all the pieces we need to calculate the particle's final weight. The particles final weight will be calculated as the

product of each measurement's Multivariate-Gaussian probability. The Multivariate-Gaussian probability has two dimensions, x and y. The "Xi" of the Multivariate-Gaussian is the i-th measurement (map coordinates) on “particle_observations”. The mean (mu-i) of the Multivariate-Gaussian is the measurement's associated landmark position (landmark associated with the Xi measurement - map coordinates) and the Multivariate-Gaussian's standard deviation (sigma) is described by our initial uncertainty in the x and y ranges. The Multivariate-Gaussian is evaluated at the point of the transformed measurement's position (particle_filter.cpp lines 296- 318). Ref: compute bivariate-gaussian https://en.wikipedia.org/wiki/Multivariate_normal_distribution#Bivariate_case

        - To finish this step we now normalize the weights (particle_filter.cpp lines 325).
    - **Re-sample**: What the following code specifically does is randomly, uniformly, sample from the cumulative distribution of the probability distribution generated by the weights . When you sample randomly over this distribution, you will select values based upon there statistical probability, and thus, on average, pick values with the higher weights (i.e. high probability of being correct given the observation z). We will later see that we will select the particle with the highest probability (confidence) as our best location estimate particle_filter.cpp lines 371-384).

4. The rest of the implementation is a set of helper functions needed, including some display features.

STEP 2: Results & Future work

Results

The problems detected from the previous submission were:

- Motion model issue and how the noise was applied.
- Main problem: Landmark association. My initial approach to narrow down the landmarks that could be associated to a specific observation (predicted_landmarks) by looking simply if it falls into the FOV of the specific particle, had a bug in the way the FOV range was calculated. For this very simple case where we have only 4 landmarks it’s easier just to go through all of them for each observation and find which landmark is the “closest” to that observation.

After solving these 2 problems, the results were “good”. It’s hard to quantify how “good” since we don’t have the robot ground truth to evaluate the particle filter’s performance with a simple Means Square Error (MSE) for example. At this point and with the time given, there are only 2 ways to measure the PF performance:

1. Simple Sim visual observation: Best Estimate location and orientation w.r.t to the robot (visual)
2. Best Particle’s Confidence (or weight).

Problems that still need to be addressed.

- The PF seems to be very sensitive to disturbances on the robot’s orientation. When we see these disturbances applied is when the PF gets “totally lost” and it’s estimations become totally unreliable until enough iterations have happened to regain accuracy/confidence. This is mainly due to the symmetry of the environment. Increasing the number of particles does not solve this problem. This is due to most of the particles converging into the “best estimate” before the

disturbance occurs and falling into the symmetry of the environment, anyway. A simple way to solve this problem is described below (increasing the number of landmarks and/or FOV)

- The combination of “reduced number of landmarks” and small sensing “Field-of-View” makes this problem challenging for the PF. Experimenting with the FOV proved to be sufficient to overcome the low number of landmarks. With 275 degrees FOV, the robot can capture, in any situation, at least 2 landmarks; making the localization very robust despite the high variance/noise.
- With as few as 25 Particles the PF can successfully localize the robot, but to recover/solve the Kidnapped problem (especially the 180 degrees orientation switch on the robot and the symmetry issue) will require 100-150 particles to recover in a relatively short time before another switch is triggered and perpetuates the “lost state” of the PF (Refer to the stdouts when running and see how the “highest confidence” gets very close to 1(or one in many cases) as we increase the number of particles.

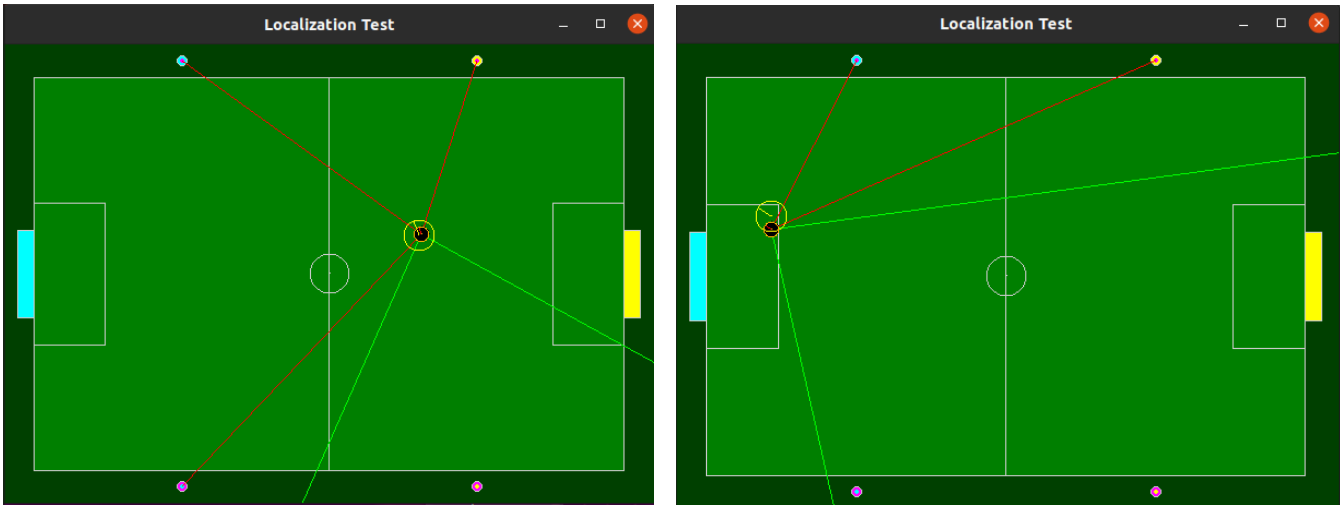


Figure 1: 25 particles with 275 degrees FOV

I included a DRAFT of my UKF in an attempt to perform a performance comparative study of both methods. Please feel free to reach out for more details on my UKF initial implementation (Eigen library not included for convenience)

Future Work

Here are come of the areas that I wanted to work in more depth but I simply didn’t have enough time:

- Computationally more efficient “Closest Neighbor” algorithms to find & associate landmarks to specific observations (on particle coordinate frame).
- More efficient “Re-sample”. Now, the code generates/re-samples all N particles based in their prior belief. This approach is ok, but we may use an “Effective Percent Threshold” and a “min prior believe threshold” so that the number of particles that DO NOT reach this minimum prior threshold, they get eliminated and only this number of new particles get re-sampled. The literature is full of smart approaches to optimize this process.
- Implement a more sophisticated “best_particle approximation” based in all particles to be able to infer (in a continuous space) the location of the best particle, rather than just choose one with the best weight (discrete approach)

- I’d like to consolidated and make clear and clean all transformations to increase readability and maintainability of the code base.
- I’d like to add color-code to the particles displayed so they can show their “confidence” level (weights). Also I’d like to add a “collect data” feature and create a few plots with this data to better understand the entire process: motion update, sensor update, landmark association, etc.
- It is highly needed to add more logging (glogs) and more checks.
- It is imperative to add unit tests to better diagnose the problems and other areas of improvement such a time and space complexity.
- I’d love to have the ground truth of the robot to develop performance studies and understand better where and why the algorithm can be improved.
- Implement a UKF:
    - Take all the observations and associate them to the closest landmark
    - Make a coordinate transformation of all observations to the coordinate frame on the landmark that is associated with. We will end up with x,y locations of the robot as if they were made by the landmarks
    - Implement a UKF with this observations.
    - In fact with the way we see the robot moving we can easily implement a KF assuming a linear motion. When the robot is kidnapped the KF will struggle to find the new location & orientation.